Polis University – Tirana (AL)

Lecture Notes on Advanced Java Programming

Luca Lezzerini – Fatjona Lorja

Rev. 1.5
28/05/2024

Table of Contents

1 Nested Classes

For details, please refer to *https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html*

In Java, a nested class is a class that is defined within another class (that is called the "outer" class, which is the "container" class, the class that contains the nested one).

Nested classes are divided into two categories: non-static and static.

Non-static nested classes are called **inner classes**. They are declared without the static keyword and can access both static and non-static members of the enclosing class.

Inner classes must be instantiated starting from an instance of the outer class. This is because they can access private instance members of the outer class and, consequently, they need an instance where to look into, as shown in the following example:

```
OuterClass outerObject = new OuterClass();

OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Nested classes that are declared *static* are called static nested classes and can be accessed using the enclosing class name, like OuterClass.NestedClass (as any other static member of a class). They can access only static members of the enclosing class and cannot access non-static members.

Local classes are classes defined inside a method. In this sense, we write the class definition directly in the middle of the method's code and we can use these classes only in that method. Local classes can access private members of the outer class and can also access local method variables but only if they are final or de facto final. This means that the local variable must be declared final or must be declared and initialized without changing its value.

1.1 Anonymous Classes

Anonymous classes are a more concise expression for local classes that implement a given interface. They enable you to declare and instantiate a class at the same time. They are local classes, but they do not need a name (in this sense are anonymous). Use them if you need to use a local class only once. Anonymous classes are usually used for implementing interfaces or extending classes in a single statement. So, this syntax allows us to implement an interface to a class with no name and immediately instantiate it.

So, nested classes in Java encapsulate related functionality and can be used to make code more organized and modular. They also provide a way of accessing private members of the enclosing class, which can be useful in certain situations. They increase encapsulation and ease more readable and maintainable code.

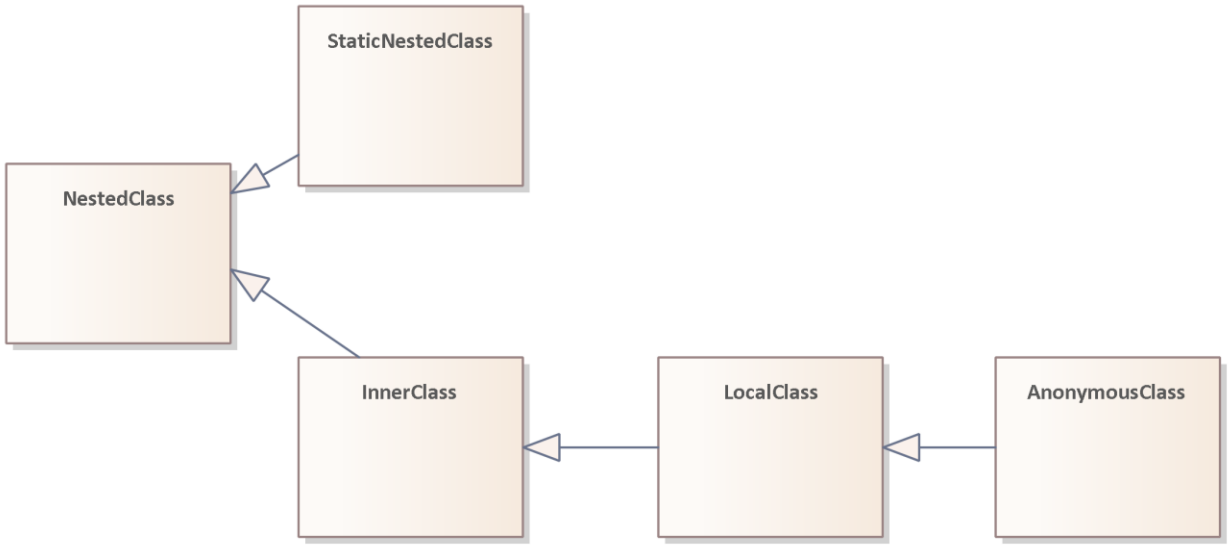


Figure 1 - Nested class hierarchy

1.2 Lambda Expressions

Lambda expressions are a concise notation to use anonymous classes in the case of an interface that has only one method (not counting, static, default and private methods). Such an interface with a unique method is called a "**functional interface**".

Lambda expressions are a concise way of representing functions or methods as objects. They were introduced in Java 8 and are a part of the functional programming paradigm in Java. They can be also considered as a way of passing code as a parameter to a method.

A lambda expression consists of three parts: the parameter list (on the left), the arrow operator ->, and the body of the function (on the right). Left and right are with reference to the arrow.

Lambda expressions can be used wherever an object of a functional interface is expected. A functional interface is an interface that has only one abstract method. Lambda expressions provide a way of implementing the abstract method of a functional interface without having to create a separate class for the implementation.

Lambda expressions provide a way of writing more concise and readable code in Java, especially when working with collections, streams, and other functional programming constructs. They help to reduce the boilerplate code that is required when implementing functional interfaces using anonymous inner classes.

Lambda expressions are very important because, if used with Generics, allow the definition of "generic algorithms" which are code that can work with almost any type to implement a wide range of behaviours, both passed (the type and the behaviour) as parameters.

In the case of a functional interface, in fact, the syntax of an anonymous class can be simplified using the lambda expression, as shown below:

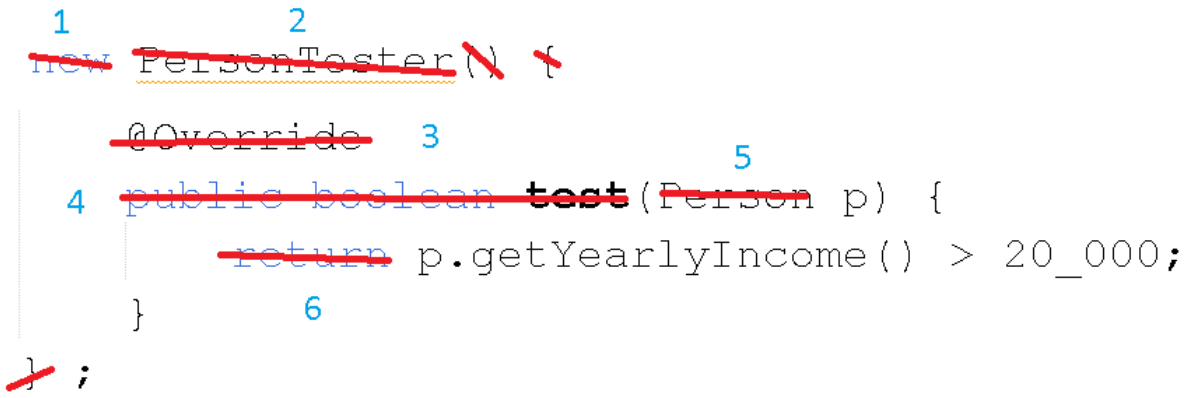


Figure 2 - Anonymous class simplified as lambda expression

In the above image the following elements are removed:

1. We want to create an object so new is obvious
2. We are putting this object in a variable or in a parameter of PersonTester type, so the type is superfluous
3. We are implementing a functional interface, so overriding is obvious
4. The functional interface has only one method so it is not needed to say which (among one) must be implemented
5. The unique method of the interface expects a Person, sono doubt that it is a Person and so no need to tell explicitly
6. The lambda expression is an expression and, consequently must have a returned value so no need of return instruction

The result of these simplifications written as a lambda expression is:

```
p -> p.getYearlyIncome() > 20_000
```

1.3 Method References

Method references in Java are a shorthand notation for referring to a method or constructor of a class. They were introduced in Java 8 and are closely related to lambda expressions.

Method references allow you to pass a method as a reference to another method, instead of writing a lambda expression to implement the method. They provide a way of reducing the verbosity of lambda expressions and can make code more concise and readable.

There are four types of method references in Java:

- Reference to a static method
- Reference to an instance method of a particular object
- Reference to an instance method of an arbitrary object of a particular type
- Reference to a constructor

2 The Stream interface

Refer to the following tutorial https://www.geeksforgeeks.org/stream-in-java/

Also refer to the following Javadoc for details:

https://docs.oracle.com/en%2Fjava%2Fjavase%2F21%2Fdocs%2Fapi%2F%2F/java.base/java/util/stream/Stream.html

3 Generics

For further details please refer to https://docs.oracle.com/javase/tutorial/java/generics/index.html

Generics add stability to your code by making more of your bugs detectable at compile time. Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces, and methods. Much like the more familiar formal parameters used in method declarations, type parameters provide a way for you to reuse the same code with different types of inputs. The difference is that the inputs to formal parameters are values, while the type parameters of generics are types.

We could achieve something similar declaring a parameter or a property of Object class. In this way we can store any object (of any class) into them. But the problem is that the Java compiler assumes the Object interface and, so, we have to continuously cast to allow the use of the right interface. But using a lot of casts with Object variables trends to provide poor code because there no control at compiling time of type coherence (cast checks at runtime). As a consequence, the code has a high risk of ClassCastException.

Generics are a way of creating type-safe classes and methods that can work with different types of objects. Generics work by allowing the programmer to specify a type parameter when declaring a class, interface, or method. This type parameter is represented by a placeholder or the wildcard character (i.e. the "?"), typically represented by the letter T, E, or K. The actual type that is used for the type parameter is specified when an object of the generic type is created or when the method is called. So, Generics in Java provide a way of writing reusable and type-safe code. They allow the programmer to specify a type parameter for a class or method without knowing the actual type that will be used until runtime. This makes it easier to write code that can work with different types of objects and reduces the risk of type errors at runtime because its formal coherence over types is checked at compile time (i.e. on my development notebook) and not at runtime (i.e. at customer's data centre when thousands of users are using it).

Why use generics:

- Stronger type checks at compile time.

- Elimination of casts.
- Enabling programmers to implement generic algorithms (with lambdas).

The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

4 Optional

For further reference: *https://www.baeldung.com/java-optional*

In Java programming language, Optional is a class introduced in Java 8 that represents a container object that may or may not contain a non-null value but that is never null. The purpose of Optional is to provide a safer and more explicit way of handling null values in Java applications.

An Optional object can be instantiated in one of two ways: with a non-null value, using the of() method, or with a null value, using the empty() method. Once an Optional object is created, you can check if it contains a value using the isPresent() method. If the Optional contains a value, you can retrieve it using the get() method.

If the Optional is empty, calling the get() method will result in a NoSuchElementException being thrown. To avoid this, you can use the orElse() method to provide a default value to return if the Optional is empty.

```
String nullName = null;

String name = Optional.ofNullable(nullName).orElse("john");
```

Alternatively, you can use the orElseGet() method to provide a supplier function that will be called to generate a default value only when the Optional is empty.

Optional can also be used to chain multiple operations that may or may not return null values. For example, instead of nesting multiple if-else statements to handle null values, you can chain multiple map() and flatMap() methods on an Optional object to perform transformations on the contained value.

Optional provides a more concise and safe way of handling null values in Java applications, helping to avoid common NullPointerException errors.

Following, a more complex example with many used operators :

```
return Optional.ofNullable(modem2)
    .map(Modem::getPrice)
    .filter(p -> p >= 10)
    .filter(p -> p <= 15)
    .isPresent();
```

5 Exceptions

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

The Java programming language uses *exceptions* to handle errors and other exceptional events.

There are 3 kinds of exceptions:

- Checked
- Unchecked (often called runtime exception)
- Error

The *checked exception*. These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from. For this reason, checked exceptions must be handled with a try-catch of with the throws instructions.

The *unchecked exception*, is sometimes called *runtime exception*. These are exceptional conditions that a well-written application should anticipate and be able to avoid and recover from, giving good design, coding and testing. For this reason, they should never happen in a stable version of the code. Unchecked exceptions can be handled with a try-catch of with the throws instructions but this is not mandatory (because in a well-written and tested application they will not happen).

The *error*. These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from. Usually, they are related to problems in the infrastructure where the application runs. Errors can be handled with a try-catch of with the throws instruction but this is not mandatory (because usually they are catastrophic and no recovery is possible).

The exceptions are organised in a hierarchy that starts from Throwable and is depicted below. Please note that unchecked exceptions are those descending from RuntimeException (included).

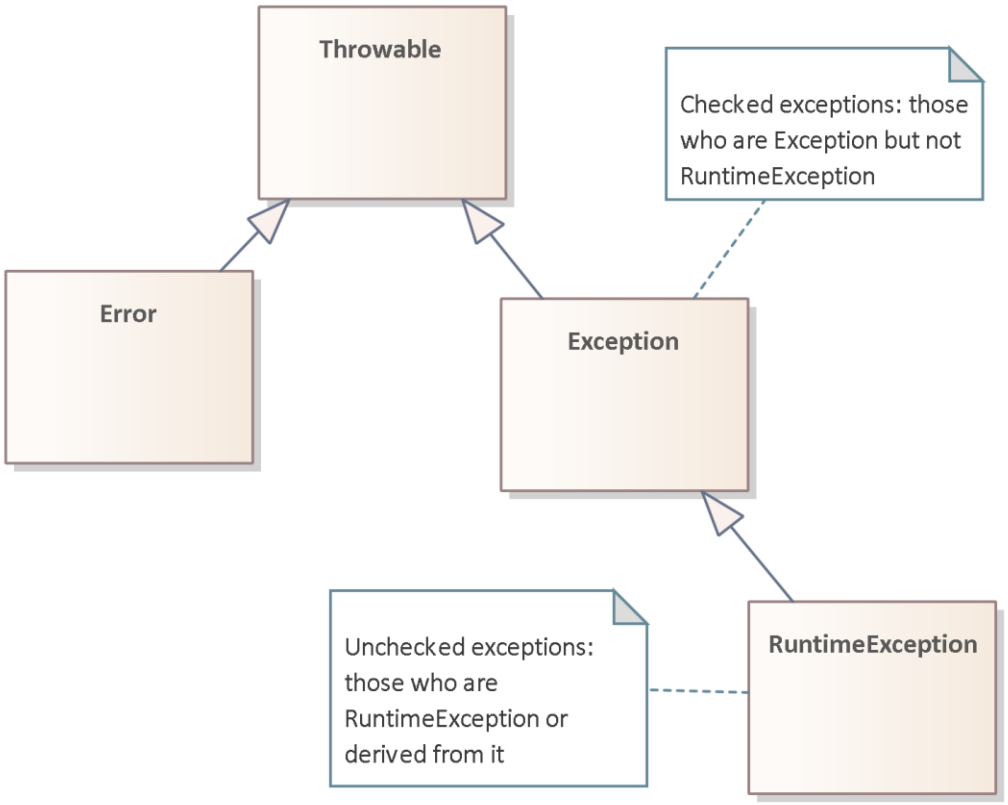


Figure 3 - Exceptions hierarchy

We can handle exceptions by using the three exception handler components — the try, catch, and finally blocks — to write an exception handler.

Exceptions can be caught and handled using a try-catch block. The basic syntax f or a try-catch block:

```
try {
    // code that may throw an exception
}
catch (ExceptionType1 e1) {
    // code to handle exception of type ExceptionType1
}
```

```
catch (ExceptionType2 e2) {
    // code to handle exception of type ExceptionType2
}
finally {
    // optional code to execute after try or catch block
}
```

In this example, the code that may throw an exception is enclosed in the try block. If an exception is thrown, the corresponding catch block is executed to handle the exception. The catch blocks are executed in the order in which they are defined, and the first block that matches the type of the thrown exception is executed. If no matching catch block is found, the exception is propagated to the calling method.

The finally block is optional and is used to specify code that should be executed regardless of whether an exception is thrown or not. This block is typically used for resource cleanup, such as closing open files or network connections.

6 Collections

A collection — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group.

The core collection interfaces encapsulate different types of collections, which are shown in the figure below. These interfaces allow collections to be manipulated independently of the details of their representation. Core collection interfaces are the foundation of the Java Collections Framework. As you can see in the following figure, the core collection interfaces form a hierarchy.

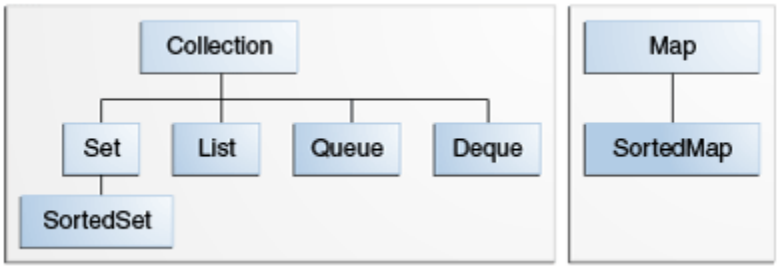


Figure 4 - Collections hierarchy

The general-purpose implementations are summarized in the following table.

Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementations
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Figure 5 - Collections implementation classes

6.1 List

The List interface is the base for collections that keep orders. This means that elements persist in a specific order (usually the insertion order or the one after a sort operation).

We typically use two kinds of implementation class of the List interface: the ArrayList and the LinkedList.

The ArrayList implements the interface RandomAccess and this means that it can be accessed (by the get method) directly, having the index position of the element with access time of O(1).

The ArrayList is a good enhancement of the array type because its size can be dynamically changed but this change is done by the ArrayList itself that starts with a predefined size and, when it needs more space, it automatically doubles its dimension. To reduce overhead due to repeated dimension changes (they are computationally expensive) the capacity parameter (i.e. the maximum expected number of elements) can be provided at creation time.

The LinkedList is another implementation of List interface but it does not implement RandomAccess so its speed when using get to access an element is very slow.

LinkedList is used if we need to quickly add and remove elements on head or queue or if we have to quickly change its structure (i.e. data sequence) without having to search inside it.

ArrayList is used if we have to add only and quick access by the index. Removal from head is very slow and also element insertion and removal.

6.2 Set

Set is a data structure that does not preserve order. It is used to store objects that can be processed or retrieved in any order. It is implemented through HashMap and TreeMap classes (through HashSet and TreeSet that are simple wrappers) and so we do not explain them because they are explained in the next section.

6.3 Map

The Map is an associative structure that does not preserve order because it associates a key to a value. It has various implementation but the two most used are TreeMap and HashMap.

Map, usually, is implemented to fast retrieve value given the key.

TreeMap uses a binary search algorithm that converges in log2(n) (where n is the size of the map) and uses only memory requested by data. The insertion and removal of elements have a cost of O(n log2(n)) due to the need to rebalance the tree used for search.

HashMap uses a hash search algorithm and its search time is O(1) if correctly designed and maintained. It uses more memory than needed due to the need to create space for all possible hash codes. The hash coding can be defined by the programmer and also the memory factor. Usually, the map occupies at least 1.3 times the needed memory, in worst cases, it can occupy many times the needed memory. It resizes and rehashes when it is near to arrive at 75% of allocated memory occupation, doubling the previous space. It does not allocate so much space but allocates hash bins, but its memory complexity is less efficient.

To avoid frequent rehashing, which slows adding elements, a sort of capacity can be given even in this case. Also, the memory factor can be provided by the programmer.

If the hash map starts colliding, its search time will be slower but, due to the use of binary search in this case, the worst search time will be O(log2(n)).

7 StringBuilder

StringBuilder in Java is a class used to create a mutable, or in other words, a modifiable succession of characters. Like StringBuffer, the StringBuilder class is an alternative to the Java Strings Class, as the Strings class provides an immutable succession of characters. However, there is one significant difference between StringBuffer and StringBuilder, and it is that the latter is non-synchronized. It means that StringBuilder in Java is a more suited choice while working with a single thread, as it will be quicker than StringBuffer.

Strings are immutable elements, so, changing even only one character causes the old string to be marked for disposal and a new one to be created with the changed character. This causes a lot of memory activity and garbage collection that results in a huge overhead.

When we need to manipulate (as a mutable) a string, we should use StringBuilder (or StringBuffer). Using the StringBuilder class will require handling a different syntax (for example the + string operator cannot be used) but the results in terms of performance will be amazing.

The following table lists and describes the constructors of StringBuilder in Java

Constructor Name	Description
StringBuilder()	It constructs a blank string builder with a capacity of 16 characters
StringBuilder(int capacity)	It creates an empty string builder with the specified capacity
StringBuilder(CharSequence seq)	It creates a string builder with the same characters specified as the argument
StringBuilder(String str)	It will construct a string builder with the string specified in the argument

The StringBuilder in Java provides numerous methods to perform different operations on the string builder. The table depicted below enlists some primary methods from the StringBuilder class.

Method	Description
StringBuilder append (String s)	This method appends the mentioned string with the existing string. You can also with arguments like boolean, char, int, double, float, etc.
StringBuilder insert (int offset, String s)	It will insert the mentioned string to the other string from the specified offset position. Like append, you can overload this method with arguments like (int, boolean), (int, int), (int, char), (int, double), (int, float), etc.
StringBuilder replace(int start, int end, String s)	It will replace the original string with the specified string from the start index till the end index.
StringBuilder delete(int start, int end)	This method will delete the string from the mentioned start index till the end index.
StringBuilder reverse()	It will reverse the string.
int capacity()	This will show the current StringBuilder capacity.
void ensureCapacity(int min)	This method ensures that the StringBuilder capacity is at least equal to the mentioned minimum.
char charAt(int index)	It will return the character at the specified index.
int length()	This method is used to return the length (total characters) of the string.
String substring(int start)	Starting from the specified index till the end, this method will return the substring.
String substring(int start, int end)	It will return the substring from the start index till the end index.
int indexOf(String str)	This method will return the index where the first instance of the specified string occurs.
int lastIndexOf(String str)	It will return the index where the specified string occurs the last.
void trimToSize()	It will attempt to reduce the size of the StringBuilder.

8 Concurrency API

8.1 Structure

The Java Concurrent API provides a framework for developing multi-threaded concurrent programs and provides a set of high-level concurrency features. Here are some of the main packages and their descriptions:

java.util.concurrent: This is the main package of Java's concurrent library. It includes a number of classes that are used for concurrent programming. Key classes and interfaces in this package include:

- ExecutorService: An interface that represents an asynchronous execution mechanism that is capable of executing tasks concurrently in the background.

- Future<V>: Represents the result of an asynchronous computation.
- ScheduledExecutorService: An ExecutorService that can schedule commands to run after a given delay, or to execute periodically.
- CountDownLatch: A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.

java.util.concurrent.atomic: This package defines classes that support atomic operations, which are operations that are performed as a single unit of work without the possibility of interference from other operations.

- AtomicInteger, AtomicLong, AtomicBoolean, etc: These classes provide atomic (i.e., thread-safe) operations on single variables, like increment and decrement operations.
- AtomicReference<V>: Provides an atomic reference to an object, which can be atomically updated.

java.util.concurrent.locks: This package provides a framework of interfaces and classes for locking and waiting for conditions more flexibly than with synchronized methods and statements.

- ReentrantLock: A reentrant mutual exclusion lock with the same basic behaviour as the implicit monitors accessed using synchronized methods and statements, but with extended capabilities.
- Condition: Factors out the Object monitor methods (wait, notify, and notifyAll) into distinct objects to allow for finer-grained locking and waiting.

java.util.concurrent.Future: The Future interface represents a future result of an asynchronous computation - a result that will eventually appear in the Future after the computation completes.

java.util.concurrent.BlockingQueue: The BlockingQueue interface represents a queue which is thread-safe to put into and take instances from. A BlockingQueue will block if you try to take while the queue is empty, or if you try to put when the queue is full.

These packages and classes are used to create and manage threads, handle inter-thread communication, synchronization, and other advanced concurrency concepts in Java. Using the concurrent API correctly can greatly simplify the development of multi-threaded applications in Java.

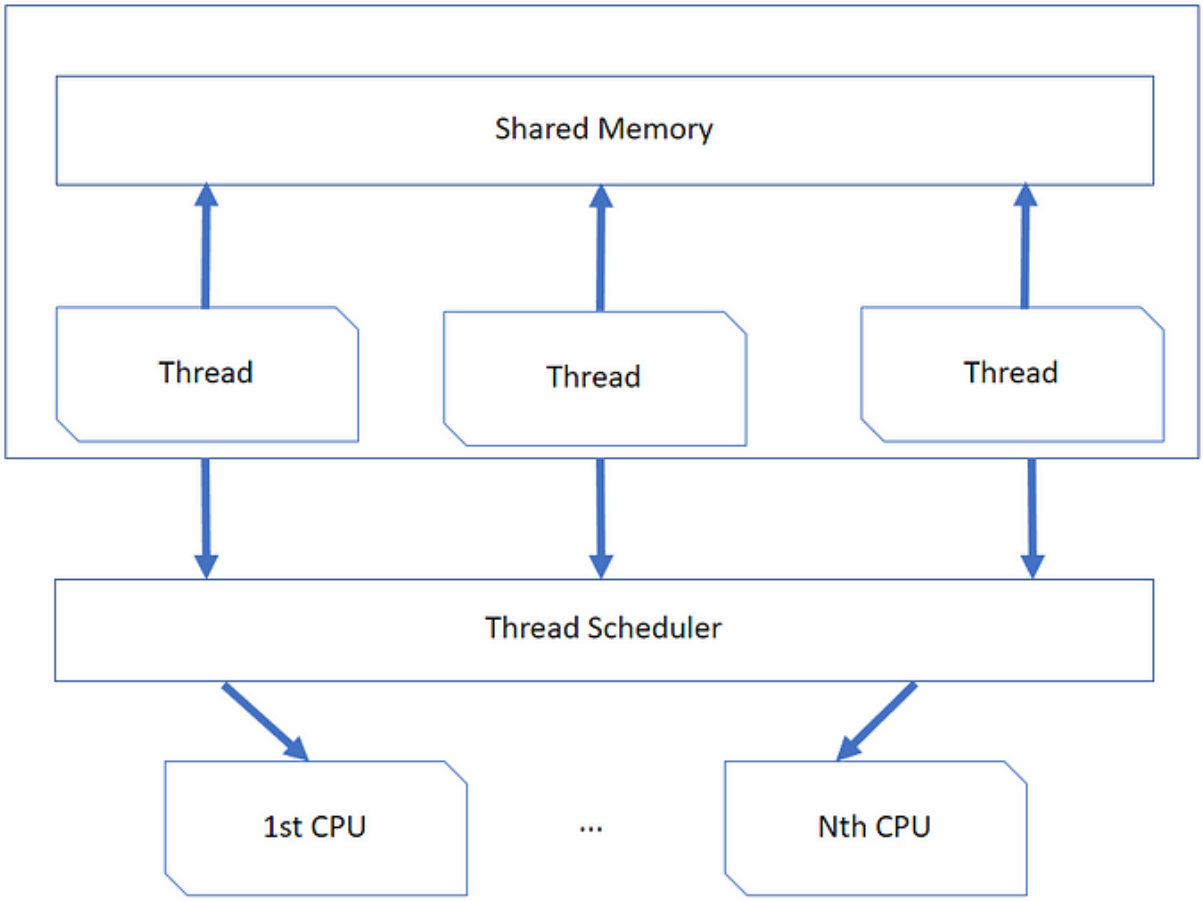
8.2 Threads

A thread means the smallest unit of execution which can be scheduled by the operating system.

A group of associated threads that execute in the same and shared environment is called a process. Single threaded process means the process that contains one thread only, multi threaded process on the other hand, means the process that contains one or more threads.

Single unit of work which is executed by thread is called as task. Even though a thread can complete multiple tasks, it can execute only one task a time.

Tabella 1 - Thread model



8.3 Concurrency

Execution of multiple processes and threads simultaneously is called as concurrency. Concurrency exists in modern programming such as multiple computers in a network, multiple applications running on one computer or multiple processors in a computer(i.e. multiple processor cores on one chip).

- Concurrency is very important in modern programming since;
- Websites must execute processes of multiple users
- Some mobile apps need to do some of the processing on servers ("in the cloud").
- Graphical user interfaces generally require processes on the background that does not interrupt the user. For example, an IDE like Eclipse compiles the Java code while the user still editing it.

As mentioned previously, multithreading can be achieved by extending Thread class.

However extending Thread class to achieve multi threading is generally not preferred. The reasons will be explained in the next section.

8.4 Runnable Functional Interface

The functional interface Runnable is commonly used to define the job that will be executed by a thread. It has one method only which is :

```
public abstract void run();
```

As mentioned in the previous section, implementing Runnable is much more preferred over extending Thread class, since:

- In Java it is possible to extend one class only, therefore extending the Thread class prevents extending other classes
- When the Thread class is extended, each thread creates a unique object and is associated with it. When the Runnable interface is implemented, it shares the same object to multiple threads.

8.5 Java Concurrency API(java.util.concurrent)

Java 5 brought new changes as Java Concurrency API to handle multi-thread programming better.

8.5.1 Executor

Executor interface is used to represent an object that executes the defined task. Executor interface has no assumption about how the task is executed. It has one method only as execute, which executes the provided task at some time in the future.

8.5.2 ExecutorService

ExecutorService is an extended interface of Executor. It goes one step beyond and provides several utility methods when compared with Executor interface. Such as shutting down thread pool or getting the result of a task.

The interface has several methods, these are the most used among them:

- **execute**: comes from Executor interface. It does the same thing.
- **submit**: submits a runnable task or value-returning task(Callable interface, which will be mentioned later on) to be executed and returns a Future Object(result of a task, which will also be mentioned later on).
- **shutdown**: triggers a shutdown, allows the previously provided tasks to be executed, and rejects any new task.
- **shutdownNow**: triggers shutdown to terminate all the previously provided but not finished tasks. It returns the list of terminated tasks.

Output of multi threaded program(using ExecutorService)

Although submit and execute seems similar, there is one important difference, submit does the exact thing execute does, however submit returns a Future Object which can be useful to track the result. It is preferred to use submit instead of execute even if the Future Object is not used since execute does not support Callable.

8.5.3 ScheduledExecutorService

This interface is similar to ExecutorService, but it can execute the tasks periodically and/or with a delay. Both Runnable and Callable can be used to define the tasks.

8.5.4 Thread Pools

The outputs of threads run concurrently with main thread but they also run concurrently with each other. However, this was changed when the section with Java Concurrency API classes comes . The reason is on the first examples a new Thread is created for each task in the first examples, however, in Java Concurrency API examples, each task run on the same thread. So each task do not work concurrent with each other but with the main thread.

The utility class Executors has various factory methods to create a pool of Threads along with the method that creates one Thread only which were used on Java Concurrency API code examples previously. The term "thread pool" here means a set of pre-instantiated reusable threads which can execute a group of arbitrary task.

A pool of threads can be created with:

- newCachedThreadPool()
- newFixedThreadPool(int nThreads)
- newScheduledThreadPool(int nThreads)

8.5.5 Future

A Future object represents the asynchronous execution result. It has various methods such as:

- **get**: waits if the task is not complete then retrieves the result
- **cancel**: cancels the execution of the task. It fails if the task is completed or cancelled already.
- **isDone**: returns true if the task is completed
- **isCanceled**: returns true if the task was cancelled before it completes

8.5.6 Callable Functional Interface

Callable is a functional interface with the method call(). Unlike the Runnable interface, it has a generic return value. The result comes wrapped with the Future.

9 Future and CompletableFuture

9.1 Future

In a few words, the Future class represents a future result of an asynchronous computation. This result will eventually appear in the Future after the processing is complete.

Long-running methods are good candidates for asynchronous processing and the Future interface because we can execute other processes while we're waiting for the task encapsulated in the Future to complete.

Some examples of operations that would leverage the async nature of Future are:

- computational intensive processes (mathematical and scientific calculations)
- manipulating large data structures (big data)
- remote method calls (downloading files, HTML scrapping, web services)

A Future represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using the method get when the computation has been completed, blocking if necessary until it is ready. Cancellation is performed by the cancel method. Additional methods are provided to determine if the task was completed normally or was cancelled. Once a computation has been completed, the computation cannot be cancelled. If you would like to use a Future for the sake of cancellability but not provide a usable result, you can declare types of the form Future<?> and return null as a result of the underlying task.

Table 1 - Example of Future usage

```
interface ArchiveSearcher { String search(String target); }
 class App {
   ExecutorService executor = ...
   ArchiveSearcher searcher = ...
   void showSearch(final String target)
       throws InterruptedException {
    Future<String> future
      = executor.submit(new Callable<String>() {
        public String call() {
            return searcher.search(target);
        }});
    displayOtherThings(); // do other things while searching
    try {
      displayText(future.get()); // use future
    } catch (ExecutionException ex) { cleanup(); return; }
   }
 }
```

Future is a generic interface that represents the result of an asynchronous computation. It has the following methods:

- boolean cancel(boolean mayInterruptIfRunning): Attempts to cancel the computation. If the computation has already completed or cannot be cancelled, this method returns false. Otherwise, it returns true.
- boolean isCancelled(): Returns true if the computation was cancelled before it completed normally.
- boolean isDone(): Returns true if the computation has completed, whether it completed normally, was cancelled, or terminated due to an exception.

- V get() throws InterruptedException, ExecutionException: Waits if necessary for the computation to complete, and then retrieves its result. If the computation was cancelled, this method throws a CancellationException. If the computation is completed due to an exception, this method throws an ExecutionException. If the current thread was interrupted while waiting for the result, this method throws an InterruptedException.
- V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException: Waits the specified amount of time for the computation to complete, and then retrieves its result. If the computation has not been completed within the specified time, this method throws a TimeoutException. Otherwise, it behaves the same as the get() method.

Table 2 - Example of Future generated using the Call interface to calculate the sum of the first 100 numbers

```
import java.util.concurrent.*;
public class Main {
   public static void main(String[] args) throws ExecutionException, InterruptedException {
      ExecutorService executor = Executors.newSingleThreadExecutor();
     Future<Integer> future = executor.submit(new Callable<Integer>() {
         @Override
         public Integer call() throws Exception {
            int sum = 0;
            for (int i = 1; i <= 100; i++) {
               sum += i;
            }
            return sum;
         }
      });
      System.out.println("Waiting for the result...");
      int result = future.get();
      System.out.println("The sum of the first 100 numbers is: " + result);
      executor.shutdown();
   }
}
```

9.2 Handling exceptions

If a Callable task throws an exception, it will be wrapped in an ExecutionException and thrown by the get() method of the Future object. You can handle this exception by using a try-catch block:

```
try {
   int result = future.get();
} catch (ExecutionException e) {
   // Handle the exception thrown by the Callable task
}
```

Alternatively, you can use the get() method that takes a timeout and a TimeUnit argument. This method will throw a TimeoutException if the computation has not been completed within the specified time.

```
try {
    int result = future.get(1, TimeUnit.SECONDS);
} catch (ExecutionException | TimeoutException e) {
    // Handle the exception thrown by the Callable task or the timeout
}
```

9.3 Cancelling Tasks

You can cancel a Future task using the cancel() method. This method returns true if the task was successfully cancelled, and false if the task could not be cancelled or had already been completed.

```
if (future.cancel(true)) {
    System.out.println("Task cancelled successfully");
} else {
    System.out.println("Task could not be cancelled");
}
```

Note that cancelling a task does not guarantee that it will stop immediately. It is up to the implementation of the Callable task to check the isCancelled() method and stop executing if necessary.

You can use the cancel() method of the Future interface to attempt to cancel the execution of a task. If the task has already started, it may not be possible to cancel it. The isCancelled() method returns true if the task was successfully cancelled, and false otherwise.

Here is an example of cancelling a Future task:

```
import java.util.concurrent.*;
public class Main {
   public static void main(String[] args) throws ExecutionException, InterruptedException {
      ExecutorService executor = Executors.newSingleThreadExecutor();
Future<String> future = executor.submit(() -> {
         try {
            Thread.sleep(1000);
            return "Hello World!";
         } catch (InterruptedException e) {
            return "Interrupted";
         }
      });
future.cancel(true);
if (future.isCancelled()) {
         System.out.println("Task was cancelled");
      } else {
         System.out.println(future.get());
      }
executor.shutdown();
   }
}
```

In this example, we use the submit() method of the ExecutorService to submit a Callable task that sleeps for 1 second and returns a string. We then use the cancel() method to attempt to cancel the task. The isCancelled() method returns true if the task was successfully cancelled, and false otherwise.

9.4 Waiting for a Future to Complete

You can use the get() method of the Future interface to block the current thread until the task completes, and then retrieve the result of the computation. The get() method can throw a checked ExecutionException if the task threw an exception, or a checked InterruptedException if the current thread was interrupted while waiting for the task to complete.

```
import java.util.concurrent.*;
public class Main {
   public static void main(String[] args) throws ExecutionException, InterruptedException {
      ExecutorService executor = Executors.newSingleThreadExecutor();
      Future<String> future = executor.submit(new Callable<String>() {
         @Override
         public String call() throws Exception {
            while (!Thread.currentThread().isInterrupted()) {
               // Do some work
            }
            return "Task cancelled";
         }
      });
      // Cancel the task after 1 second
      Thread.sleep(1000);
      future.cancel(true);
      System.out.println("Waiting for the result...");
      String result = future.get();
      System.out.println(result);
      executor.shutdown();
   }
}
```

You can also use the get(long timeout, TimeUnit unit) method to specify a timeout

9.5 Handling Timeouts

You can use the get(long timeout, TimeUnit unit) method of the Future interface to specify a timeout and a TimeUnit for waiting for the task to complete. If the task is not complete within the specified timeout, the get() method will throw a checked TimeoutException.

Here is an example of using the get(long timeout, TimeUnit unit) method to handle a timeout:

```
import java.util.concurrent.*;
public class Main {
   public static void main(String[] args) throws ExecutionException, InterruptedException {
      ExecutorService executor = Executors.newSingleThreadExecutor();
Future<String> future = executor.submit(() -> {
         try {
            Thread.sleep(2000);
            return "Hello World!";
         } catch (InterruptedException e) {
            return "Interrupted";
         }
      });
try {
         String result = future.get(1, TimeUnit.SECONDS);
         System.out.println(result);
      } catch (TimeoutException e) {
         System.out.println("Timeout");
      }
executor.shutdown();
   }
}
```

In this example, we use the submit() method of the ExecutorService to submit a Callable task that sleeps for 2 seconds and returns a string. We then use the get(long timeout, TimeUnit unit) method to specify a timeout of 1 second and a `TimeUnit'.

9.6 Chaining Futures

You can use the thenApply() method of the CompletableFuture class to chain multiple Future tasks together. This method takes a Function argument, which is applied to the result of the previous Future task, and returns a new CompletableFuture that completes with the result of the function.

```
import java.util.concurrent.*;
public class Main {
   public static void main(String[] args) throws ExecutionException, InterruptedException {
      ExecutorService executor = Executors.newFixedThreadPool(2);
      CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> {
         try {
            Thread.sleep(1000);
            return "Hello";
         } catch (InterruptedException e) {
            return "Interrupted";
         }
      }, executor);
      CompletableFuture<String> future2 = future1.thenApplyAsync(s -> s + " World!",
executor);
      CompletableFuture<String> future3 = future2.thenApplyAsync(s -> s + "!", executor);
      System.out.println(future3.get()); // "Hello World!"
      executor.shutdown();
   }
}
```

In this example, we use the supplyAsync() method of the CompletableFuture class to create an asynchronous task that sleeps for 1 second and returns a string. We then chain three Future tasks together using the thenApplyAsync() method, which applies a function to the result of the previous task and returns a new CompletableFuture.

The thenApplyAsync() method takes an Executor argument, which specifies the executor that will be used to execute the task. In this case, we use the same ExecutorService for all tasks.

Finally, we use the get() method of the last Future task to retrieve the result of the computation.

9.7 Asynchronous Callbacks

You can use the thenAccept() method of the CompletableFuture class to specify a callback that will be executed when the Future task completes. This method takes a Consumer argument, which is applied to the result of the Future task.

Here is an example of using the thenAccept() method to print the result of a Future task:

```
import java.util.concurrent.*;
public class Main {
   public static void main(String[] args) throws ExecutionException, InterruptedException {
      ExecutorService executor = Executors.newSingleThreadExecutor();
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
         try {
            Thread.sleep(1000);
            return "Hello World!";
         } catch (InterruptedException e) {
            return "Interrupted";
         }
      }, executor);
      future.thenAccept(System.out::println);
      executor.shutdown();
   }
```

In this example, we use the supplyAsync() method to create an asynchronous task that sleeps for 1 second and returns a string. We then use the thenAccept() method to specify a callback that will print the result of the task.

Note that the thenAccept() method does not return a new CompletableFuture, so it cannot be used to chain multiple tasks together.

9.8 Combining Futures

You can use the thenCombine() method of the CompletableFuture class to combine the results of two Future tasks. This method takes another CompletableFuture and a BiFunction argument, which is applied to the results of both Future tasks, and returns a new CompletableFuture that completes with the result of the function.

Here is an example of using the thenCombine() method to compute the sum of two Future tasks:

```
import java.util.concurrent.*;

public class Main {
   public static void main(String[] args) throws ExecutionException, InterruptedException {
      ExecutorService executor = Executors.newFixedThreadPool(2);
CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() -> {
         try {
            Thread.sleep(1000);
            return 10;
         } catch (InterruptedException e) {
            return 0;
         }
      }, executor);
      CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> {
         try {
            Thread.sleep(2000);
            return 20;
         } catch (InterruptedException e) {
            return 0;
         }
      }, executor);
      CompletableFuture<Integer> future3 = future1.thenCombine(future2, (x, y) -> x + y);
      System.out.println(future3.get()); // 30
      executor.shutdown();
   }
}
```

In this example, we use the supplyAsync() method to create two asynchronous tasks that sleep

9.9 Handling Exceptions in Callbacks

You can use the exceptionally() method of the CompletableFuture class to specify a callback that will be executed if the Future task completes exceptionally (i.e. if it throws an exception). This method takes a Function argument, which is applied to the exception thrown by the task, and returns a default value to be used as the result of the CompletableFuture.

Here is an example of using the exceptionally() method to handle an exception thrown by a Future task:

```
import java.util.concurrent.*;
public class Main {
   public static void main(String[] args) throws ExecutionException, InterruptedException {
      ExecutorService executor = Executors.newSingleThreadExecutor();
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
         throw new RuntimeException("Something went wrong");
      }, executor);
      CompletableFuture<String> future2 = future.exceptionally(ex -> {
         System.out.println("Exception: " + ex.getMessage());
         return "Default value";
      });
      System.out.println(future2.get()); // "Default value"
      executor.shutdown();
   }
}
```

In this example, we use the supplyAsync() method to create an asynchronous task that throws an exception. We then use the exceptionally() method to specify a callback that will be executed if the task throws an exception. The callback prints the exception message and returns a default value to be used as the result of the CompletableFuture.

9.10 Waiting for Multiple Futures

You can use the allOf() method of the CompletableFuture class to create a new CompletableFuture that will be completed when all of the specified Future tasks have been completed. This method takes an array of CompletableFuture objects and returns a new CompletableFuture that completes with the results of all of the tasks.

Here is an example of using the allOf() method to wait for multiple Future tasks:

```
import java.util.concurrent.*;

public class Main {
   public static void main(String[] args) throws ExecutionException, InterruptedException {
      ExecutorService executor = Executors.newFixedThreadPool(3);
      CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> {
         try {
            Thread.sleep(1000);
            return "Task 1";
         } catch (InterruptedException e) {
            return "Interrupted";
         }
      }, executor);
      CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() -> {
         try {
            Thread.sleep(2000);
            return "Task 2";
         } catch (InterruptedException e) {
            return "Interrupted";
         }
      }, executor);
      CompletableFuture<String> future3 = CompletableFuture.supplyAsync(() -> {
         try {
            Thread.sleep(3000);
            return "Task 3";
         } catch (InterruptedException e) {
            return "Interrupted";
         }
      }, executor);
      CompletableFuture<Void> allTasks = CompletableFuture.allOf(future1, future2, future3);
      allTasks.get(); // Wait for all tasks to complete
      System.out.println(future1.get()); // "Task 1"
      System.out.println(future2.get()); // "Task 2"
      System.out.println(future3.get()); // "Task 3"
      executor.shutdown();
   }
}
```

In this example, we use the supplyAsync() method to create three asynchronous tasks that sleep for different amounts of time and return a string. We then use the allOf() method to create a new CompletableFuture that will be completed when all of the tasks have been completed.

We use the get() method of the allTasks CompletableFuture to wait for all tasks to be completed. Then, we use the get() method of each Future task to retrieve the result of the computation.

10 Unit Testing

10.1 Automated testing definition

Before we start with the automation tests, let us first understand what the term 'automation' means. Automation is a process by which we can automate a manual process with the use of technology. The goal is to eliminate or reduce human and manual effort. Now let us see how automation helps in software testing (test automation).

Software testing includes writing test cases once and then executing them over and over again, when necessary.

Test execution, if done manually, is a tedious and time-consuming task. Test automation helps reduce test execution time as test scripts written once can be automatically executed any number of times without any human intervention.

10.2 What is test automation?

Test automation is a type of software testing that involves the execution of automated test cases using an automation tool.

Thus, it automates the manual testing process. The software tester writes test scripts and then executes them on demand or by scheduling them for periodic executions. This reduces the overall testing time, thus helping to release products more quickly.

10.3 What to automate

Now that we know what exactly test automation is, let's check which test cases to automate or which are ideal candidates for automation.

- Test cases that test the critical functionality of the application: for example for an e-commerce application, the critical functionality would be discovering the product via search and category pages, adding it to the shopping cart and then purchasing the functionality. Therefore, these test cases should be chosen first. The test cases for adding to the wish list and notifications, etc., should be chosen first. They should have a lower priority and therefore chosen accordingly for automation.
- Test cases that require repeated testing with a large data set: there are many test cases or application flows that require the execution of an action repeatedly. These test cases are also ideal candidates for automation as, once automated, a considerable amount of test work is reduced.
- Let us take an example of a search function of an application. We can automate the search flow with a search term and then test the search results. Then the same script can be run over and over again with a different type of search term such as single word, multi-word, alphanumeric, special characters, foreign language characters, etc.
- Time-consuming tests: workflows that require a considerable amount of time for execution and configuration should also be ideal candidates for automation.
- Take the example of the e-commerce application, if some test cases require the configuration of several products and then the execution of certain operations on those

products. Such test cases, when automated, not only reduce test execution time but also free manual testers from redundant tasks and help them focus on other exploratory testing activities.

- Test cases that need to be executed in a parallel or distributed environment - Some test cases require simultaneous operations to be performed on the application, for instance in the case of performance testing or scenarios where it is necessary to check the application's behaviour when accessing resources simultaneously. In these cases, manual testing is not feasible or would require many more resources to test particular scenarios. These automated scripts help by making simultaneous requests and collecting the results in one place.

10.4 What not to automate

It is also important to understand what kind of test cases cannot or rather should not be automated.

- Test cases related to the user interface: test cases related to the graphical user interface should be left to manual testing or human validation. This is because even with the slightest change in the user interface, test cases would fail and it is also very difficult to create reliable user interface test cases across multiple devices and screen resolutions. Anyway, there is a type of test automation that allows this kind of test.
- Usability-related test cases - Rather than 'should not', it is the case that 'cannot' automate. Usability-related test cases - ease of use of the application by different user groups that cannot be automated with current technology.
- Features that are rarely used and take time for scripting: it is good to automate complex scenarios, but investing one's efforts in scenarios that would rarely be used does not provide a good return on investment.
- Exploratory tests: exploratory tests require immediate learning of the application and simultaneous testing. Therefore, exploratory testing scenarios cannot be automated.

10.5 When should we automate

Once all the functionalities of the automation suite have been defined during test planning, we can start the task of creating the automation framework in parallel with the development team. However, the scripting of test cases should be done at the right time.

For better ROI (Return On Investment) automation and to avoid any rework, test case scripting should be started when the application is stable and no frequent changes are expected in the application.

The following are some of the advantages or benefits of test automation:

- Test automation reduces the overall test execution time. Since automated test execution is faster than manual test execution.
- It reduces the cost and resource requirements of the project, as the script created once can be made to run any number of times as long as there is no change in demand.
- It helps to work with a large set of inputs that is not feasible with manual testing.

- It helps to create a continuous integration environment where, after each code input, the test suite is automatically executed with the new build. Using CICD tools such as Jenkins, we can create jobs that execute test cases after a build is deployed and send the test results to interested parties.

10.6 Junit 5

JUnit 5 is the current generation of the JUnit testing framework, which provides a modern foundation for developer-side testing on the JVM. This includes focusing on Java 8 and above, as well as enabling many different styles of testing.

JUnit is the most popular Java unit testing framework. An open-source framework, it's used to write and run repeatable automated tests.

11 Spring Architecture

In the following diagram, the Spring architecture is depicted.

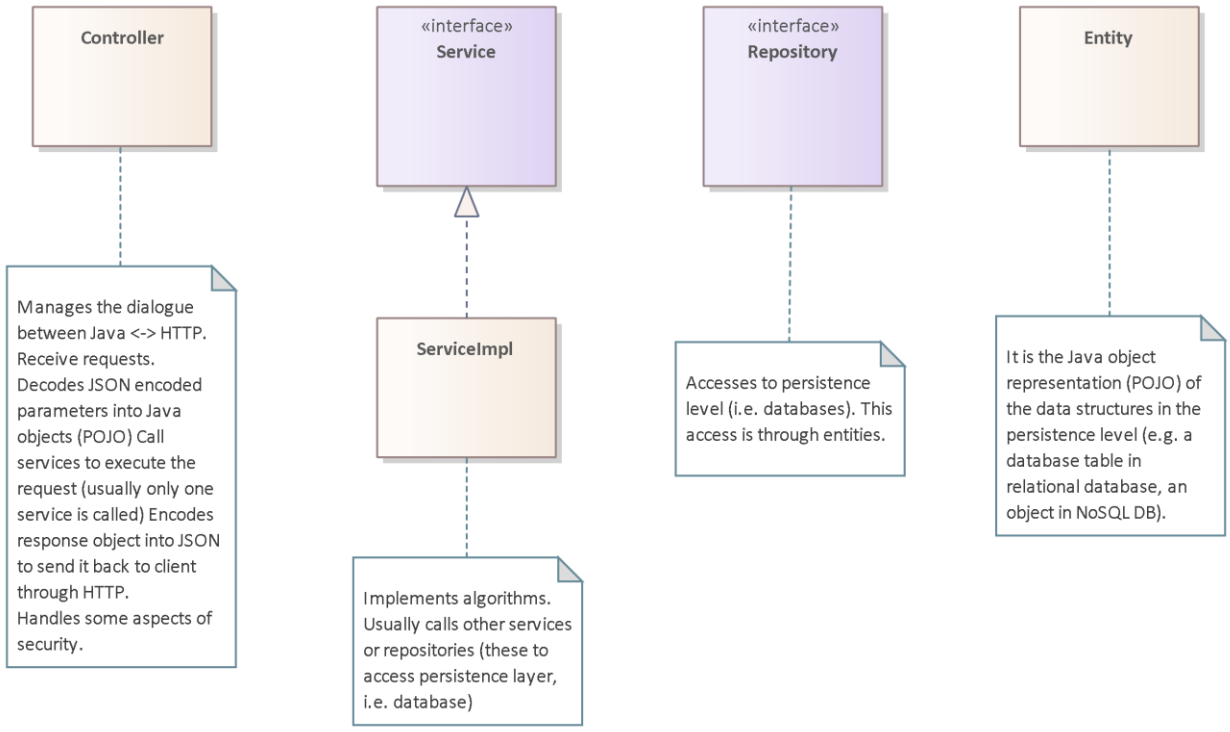


Figure 6 - Class diagram of Spring architecture

Controller: A Controller is a component that handles incoming requests from the user and sends the response back. It acts as an interface between the user and the system. So, it is an interface for the client. Controllers are responsible for processing user input, validating input data, and invoking the appropriate Service to process the request.

Service: A Service is a component that provides business logic and performs the necessary data manipulation required for a given operation. Services are responsible for performing complex operations, handling transactions, and coordinating multiple Repository calls. Services are structured as the Service interface and its implementation class.

A ServiceImpl is a concrete class that implements the Service interface. ServiceImpl contains the actual implementation of the business logic that the Service provides.

Repository: A repository is an interface that is responsible for querying and manipulating data. So, we use the Repository to dialogue with the database.

It is a component that provides an abstraction layer over the data persistence layer. Repositories are responsible for managing the data in the database, performing CRUD operations, and providing search and filtering capabilities. Repositories can be used to implement other queries through JPA (Java Persistence API).

Entity: An Entity is a plain Java object that represents a table in the database. The entity has only the responsibility to keep the data. Entities contain fields that map to columns in the table and methods to perform operations on the data. They are also used in NoSQL databases where there is no concept of table.

DTO: DTO (Data Transfer Object) is a design pattern used to transfer data between layers of an application. DTOs are plain Java objects that contain only the necessary data required for a given operation. DTOs are used to decouple the layers of an application and provide a flexible way of transferring data between them.

DTOs are used to solve the gap between the database layer (that has to manage and search a lot of data) and the client layer (that wants to manage only data related to the use case it is processing). DTOs are containers for data that transport only data needed by the use case according to the client's point of view. They are usually mapped from and to entities at the service level.

All these components communicate according to the following sequence diagram:

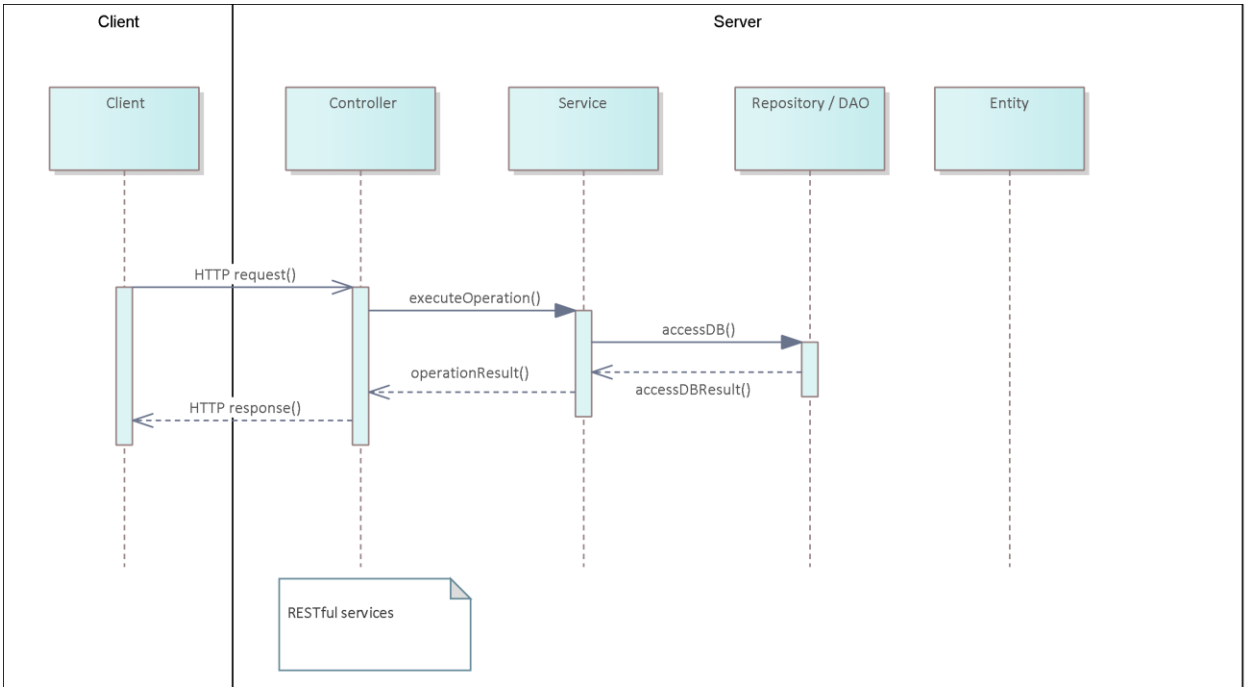


Figure 7 - Spring request sequence diagram

12 DB access in Spring: JPA, JPQL, Relationships

12.1 JPA

In Spring, there are several ways to access a database. The most used is JPA, Java Persistence API- In JPA we find many elements like Entities, the Java Persistence Query Language (JPQL), and various types of relationships between entities.

JPA: JPA is a Java standard API that provides a framework for mapping Java objects to relational database tables and to manage them abstracting from database. JPA has been defined to abstract also from ORM (Object Relational Mapper) the component used to abstract from DB technology.

JPA is used to persist data between Java object and database, (for both cases relational and NoSQL, with some adjustments in the NoSQL case). It provides a set of annotations that can be used to map entities to database tables and columns, and to design relationships among them. It provides a specific query language for relational DB that is called JPQL.

With JPA, developers can easily persist and retrieve entities from the database without writing any SQL code.

JPQL: JPQL is a query language used to retrieve data from a database using JPA. It is similar to SQL, but it operates on entities and their properties instead of tables and columns. It is used to create queries against entities to store in a relational database. JPQL queries can be used to select, filter, and order entities based on their properties.

JPQL has been designed to be object-oriented and solves most of the issues deriving from the fact that relational DB and SQL are not object-oriented.

12.2 Relationships

Relationships: In Spring, there are several types of relationships that can be established between entities in the database, including OneToMany, ManyToOne, and ManyToMany.

OneToMany: A OneToMany relationship is used when one entity can be associated with multiple instances of another entity. For example, a company can have multiple employees, and each employee is associated with one company.

ManyToOne: A ManyToOne relationship is used when multiple instances of one entity can be associated with a single instance of another entity. For example, multiple employees can be associated with a single department.

ManyToMany: A ManyToMany relationship is used when multiple instances of one entity can be associated with multiple instances of another entity. For example, multiple students can be enrolled in multiple courses.

JPA and JPQL provide powerful tools for accessing and manipulating data in a database, while relationships between entities allow developers to create complex data models and build applications that can handle a wide range of use cases.

13 Spring dependency injection

13.1 @Autowired

In Spring, Dependency Injection is a technique for managing dependencies between objects.

It allows developers to write loosely coupled code by providing an interface for defining dependencies, rather than hard-coding them into the code.

The @Autowired annotation is one of the core components of Dependency Injection in Spring. It is used to automatically wire dependencies into a bean, allowing Spring to automatically detect and inject the necessary dependencies into the bean at runtime.

With auto wiring, Spring satisfies the inversion-of-control pattern that states that is a framework's task to create the framework's objects and not a programmer's. This is one element of aspect-oriented programming that is an evolution of the object-oriented.

So @Autowired is a powerful tool for managing dependencies in Spring, allowing developers to write cleaner, more modular code that is easier to test and maintain.

13.2 Constructor injection

Although @Autowired is very often used, it can cause some issues with automated testing or in other cases. For this reason, the most recent approach to dependency injection has moved towards constructor injection.

In constructor injection, the components to be injected are yet represented by class properties but with no @Autowired annotation. They are injected passing them as parameters into the constructor and there assigned to the properties.

Tabella 2 - Example of constructor injection

```
@Service
public class MessagingServiceImpl implements MessagingService {

   private final JavaMailSender emailSender;
   private final EmailMessageQueueRepository emailMessageQueueRepository;
   private final PriorityEmailMessageQueueRepository
priorityEmailMessageQueueRepository;

   public MessagingServiceImpl(JavaMailSender emailSender,
EmailMessageQueueRepository emailMessageQueueRepository,
                        PriorityEmailMessageQueueRepository
priorityEmailMessageQueueRepository) {
      this.emailSender = emailSender;
      this.emailMessageQueueRepository = emailMessageQueueRepository;
      this.priorityEmailMessageQueueRepository = priorityEmailMessageQueueRepository;
   }
```

Often, this is done through Lombok with the @AllArgsConstructor annotation.

Tabella 3 - Example with Lombok

```
@AllArgsConstructor
public class TestController {

   private MessagingService messagingService;
   private PrincipalService principalService;
   private TestService testService;
```

14 Sample Exam Questions

14.1 First Part (intermediate exam and final exam)

- Explain what are nested classes, describing in detail: static nested classes, inner classes, local classes and anonymous classes. Write also some sample code to explain how to instantiate all the classes cited above.
- Explain in detail what are lambda expressions, making some examples of applications using the functional interfaces Predicate, Consumer and Function.

- Describe the Stream interface and its methods: stream(), parallelstream(), filter(), forEach(), map() making some examples of applications.
- Explain in detail what are method references and give some examples of usage, one for each possible type (static, instance, …).
- Explain what are generic types, and which advantages come with their usage and make some examples of the use of Generics.
- Explain the role of the wildcard (?) in generics and the unbounded, upper-bounded and lower-bounded cases, making some examples. Explain the meaning of upper bounded and lower bounded (i.e. why we choose one or the other, which is the scope of each one). Which interface can we use if we choose an unbounded generic type?
- Describe what the Optional class is, why it has been introduced in Java and make some examples of its application using the methods isPresent, filter, map and orElse. Explain how to create an Optional instance given an already instantiated object stored in the variable named 'obj'. What happens, in the Optional, if the get method is invoked with a null value stored inside the Optional instance?
- Explain in detail what are exceptions in Java. Describe the try-catch-finally statement in all its components and each one in detail. Describe what are checked exceptions and unchecked exceptions. Describe the difference between Throwable, Error and Exception classes. Describe the role of RuntimeException. Which kinds of exceptions are usually managed by Java code? Why?
- Describe in detail the three interfaces that represent the three types of collections in Java (List, Set and Map). Describe at least two implementing classes of these collection interfaces. Make an example of each of these implementing classes and describe the differences between them.
- Explain the Spring architecture and draw a sequence diagram that shows the dialogue between the various elements of the framework. For each element describe in detail its scope and draw a complete class diagram with all the typical interfaces and classes of the Spring architecture. Explain the role of the DTO and why we use it.

14.2 Second Part (final exam only)

- Explain in detail the StringBuilder class and its API, describing each method and its function. Explain when it should be used and why.
- Explain the concurrency API detailing its structure (packages). Describe in detail the Runnable and Callable interfaces, giving one example of code for each one.
- Explain the concurrency API detailing its structure (packages). Describe in detail the Future interface, giving some example of code.
- Explain the concurrency API detailing its structure (packages). Describe in detail the executors, the thread model used in Java and the three thread pools, giving at least one example of code about one executor service.
- Explain the concurrency API detailing its structure (packages). Describe in detail the chaining mechanism of Future, especially thenApply, supplyAsync, thenApplyAsync, giving one example of code for each one.

- Explain in detail what is test automation and its advantages, when to use it and when not, give a short description of Junit 5
- Explain in detail what are JPA and JPQL and how relationships are managed in Spring JPA
- Explain in detail what is dependency injection, why it is needed and the two types of it (autowiring and constructor injection)

- END OF THE DOCUMENT -